

Food Ordering App

App Resources & Setup

- Create the base resources of the app by executing the following Azure CLI script: [create-images.azcli](#). It creates the following resources:
 - resource group
 - key vault
 - managed identity
 - container registry
 - container app environment
 - storage account
- To create the container images execute: [create-images.azcli](#)

Designing & Implementing Cloud Native Applications using Microsoft Azure

Das Seminar richtet sich an Azure Entwickler und Software Architects, welche einen Überblick über die Kernelemente der Entwicklung und bereitstellung von Cloud Native Applications in Microsoft Azure bekommen wollen.

Begleitend zu den Theorieteilern der einzelnen Module, modernisieren wir eine App bestehend aus klassischem Monolithen mit UI in eine Cloud Native App mit Microservices (Catalog, Shop, State, Payment, Delivery) und Micro Frontends. Dabei legen wir Wert auf die Verwendung von Best Practices und Cloud Design Patterns, sowie deren Abbildung mit Software Architektur Diagrammen.

Wir vermitteln die Container Essentials, und Konzepte wie Stateful Containers oder SideCar Pattern und besprechen im Detail mögliche Refactorings bezüglich Bereitstellung in den Kubernetes basierenden Azure Container Apps und behandeln dabei Themen wie Secrets, Revisions, Config Injection, Health Checks, Kubernetes Event Driven Auto-Scaling - KEDA.

Dem Prinzip von Domain Driven Design folgend, vermitteln wir die Vorteile von NoSQL Datenbanken und begleiten Sie auf Ihrem Weg von Relational DB Design zum Cosmos DB NoSQL Api. Dabei behandeln wir auch die Themen Change Feed, Event Sourcing und CQRS.

Wir vermitteln die Grundlagen von Event Driven Applications, deren Transaktionsmustern, die wir mittels Saga Pattern implementieren und verbinden die einzelnen Services mittels Distributed Application Runtime (Dapr).

Wir nutzen Durable Functions, um Microservices zu implementieren, welche wahlweise Serverless aber auch also Container gehostet werden können. Im Speziellen gehen wir hier auf die Themen Durable Entities, Durable Monitoring und Durable Saga Pattern ein.

Last but not least publizieren und sichern wir die App, und deren Microservices mit API Management und Application Gateway, um dann noch unser Reactive Angular UI mit Client Side State in Echtzeit mittels Azure Web PubSub aktuell zu halten.

In allen Phasen wird Authentication und Authorization mittels Microsoft Identity sichergestellt und ein automatisiertes Deployment der App ist mittels Azure CLI und / oder BICEP gewährleistet.

Beispiele werden größtenteils in .NET, Angular und React implementiert. Fallweise können aber auch alternative Technologie Stacks (Spring Boot) verwendet werden, bzw. wird auf deren Docs verwiesen.

Voraussetzungen und Zielgruppe

Kursteilnehmer, welche die Labs erfolgreich durchführen wollen, sollten Kenntnisse und Erfahrung der in AZ-204 vermittelten Kenntnisse erworben haben. Mit RECAP gekennzeichnete Themen sind Kurzzusammenfassungen von AZ-204 Inhalten als Refresher. DevSecOps relevante Themen werden in einem separatem Kurs behandelt.

Audience: Azure Developers & Software Architects

Themen

- Introduction to Cloud Native Applications
- Building Blocks & Architecture Overview
- Container Essentials & Configuration Management
- Introduction to Azure Container Apps (ACA)
- Implementing Microservices using Azure Functions
- NoSQL Data storage using Cosmos DB
- Designing & Implementing Event Driven Apps
- Using Distributed Application Runtime - Dapr
- Optimizing and Securing API Access using Api Management
- Implementing Real Time Micro-Frontends

Introduction to Cloud Native Applications

- What are Cloud Native Applications
- App Monolith vs Microservices
- What are Cloud Architecture Design Patterns
- Microservices Communication Patterns (Sync, Async, Event Driven)
- Api Gateway Pattern, Frontend Aggregation Pattern
- What are Event Driven Applications

Building Blocks & Architecture Overview

- Food App - food ordering and delivery application
- Hosting: Containers, Kubernetes and Functions (Serverless / Containers)
- Storage: Azure Cosmos DB, Azure SQL, Blob Storage
- Configuration Management, Secrets: Key Vault, App Config Service
- Messaging Brokers: Service Bus, Event Hub, Event Grid
- Real Time: Azure SignalR Service, Azure Web PubSub
- Access & Management: API Management & Application Gateway
- Authentication & Authorization: Microsoft Identity & Managed Identities
- Provisioning base Ressourcen using Azure CLI & Bicep

Container Essentials & Configuration Management

- Container Recap (Multistage Build, Run, Debug, Publish to ACR)
- Docker Development Workflow and Debugging
- Using docker-compose.yaml to locally test multiple containers
- Configuration Management using Environment Variables, Secrets and Azure App Config Service
- Stateful Containers using Azure Blob Storage and Volume Mounts

- Understanding and using Sidecar Pattern

Introduction to Azure Container Apps (ACA)

- What is Azure Container Apps
- Azure Container Hosts: Azure Container Apps vs Kubernetes
- Deploying a multi-container App (Ingress, Egress)
- Working with Secrets & Revisions
- Using Managed Identities & Service Connectors to access services
- Using Azure App Configuration in Azure Container Apps
- Health Probes, Monitoring, Logging & Observability
- Introduction to Scaling & KEDA (Kubernetes Event Driven Auto-Scaling)
- Authentication and Authorization using Microsoft Identity Platform

Implementing Microservices using Azure Functions

- OData and Open API Support
- Hosting: Serverless vs Containers
- Hosting and Scaling containerized Functions
- Managed Identities, Key Vault and App Configuration
- Dependency Injection and Data Access using EF Core
- Durable Functions and Patterns
- Monitoring Durable Functions
- Azure Durable Entities & Actors

NoSQL Data storage using Cosmos DB

- From Relational to NoSQL: Does and Don'ts
- Domain Driven Design (DDD) and Bounded Context Pattern
- Optimize Partitioning & Performance
- Using SDKs and Entity Framework
- Cosmos DB Change Feed and Event Sourcing
- Understanding the CQRS Pattern

Designing & Implementing Event Driven Apps

- Introduction to Event Driven Architecture
- Common Message Broker Types in Azure
- Messages vs Events & Queues vs Topics
- Common Cloud Design Patterns used with Event Driven Architecture
- Publishing & subscribing messages with Cloudevents
- Event Sourcing and Integration Events
- Implementing a Saga Pattern using Durable Functions
- Orchestration vs Choreography
- Debugging Event Driven Applications

Using Distributed Application Runtime - Dapr

- Introduction to Dapr
- Understanding Dapr Architecture & Building Blocks
- Environment Setup, Debugging & State Management
- Using Dapr Components in Azure Container Apps
- Secrets and Configuration
- Publish & subscribe
- Service Invocation & Bindings

- Observability and Distributed Tracing
- Introduction to Actors

Optimizing and Securing API Access using Api Management

- API Management (APIM) Recap
- Understanding Gateway Pattern and Backends for Frontends Pattern
- API Versions and Revisions
- Securing API Access using Authentication & Managed Identities

Implementing Real Time Micro-Frontends

- Introduction to Micro Frontends
- Real Time Options: SignalR vs Azure Web PubSub
- Implementing Reactive Real Time Frontends using Event Grid & Azure Web PubSub
- Implementing a Micro Frontend as Teams App.

Lab VM Setup Guide

- Install Software
- Install Windows Subsystem Linux - Optional
- Docker Support
- Create Lab VM - Optional

Install Software

To install Software run the script `setup-az-204.ps1` from an elevated PowerShell prompt:



```
Set-ExecutionPolicy Bypass -Scope Process -Force;  
Invoke-Expression ((New-Object  
System.Net.WebClient).DownloadString('https://raw.githubusercontent.com/ARambazamba/AZ  
204/master/Setup/setup-az-204.ps1'))
```

Note: This script will run for approx 20 min. No need to wait! In the meantime you can continue to fork and clone my repo as described in the next section.

Congratulations you have completed the base setup of your class vm.

Install Windows Subsystem Linux - Optional

Requires Windows 10 - May 2020 Update or higher. To Update use this [link](#).

[Install WSL 2](#)

```
wsl --install
```

Frameworks & Runtimes

[Introduction to Bash Scripting](#)

Node

Install Node 14.x on WSL

```
sudo apt update
sudo curl -sL https://deb.nodesource.com/setup_14.x | sudo bash
sudo apt-get install -y nodejs
```

.NET 6

Register Packages:

```
wget https://packages.microsoft.com/config/ubuntu/20.10/packages-microsoft-prod.deb -O
packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
```

Install .NET:

```
sudo apt-get update; \
sudo apt-get install -y apt-transport-https && \
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-6.0
```

Azure CLI

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
az config set extension.use_dynamic_install=yes
```

Azure Function Core Tools V4

```
sudo npm install -g azure-functions-core-tools@4 --unsafe-perm true
```

Docker Support

Download & Install [Docker Desktop for Windows](#)

Configure Docker Desktop:

 docker-desktop

Create an account at [Docker Hub](#) and signin to Docker

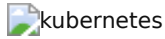
 docker-signin

Configure Docker:

 wsl-engine

 wsl-engine-resources

Enable Kubernetes:



Press Apply & Restart to complete Docker Setup

Test Installation

In the console window execute:

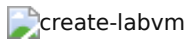
```
docker run hello-world
```



Create Lab VM - Optional

Execute `create-lab-vm.azcli` or run the following remote script in Cloud Shell

```
curl https://raw.githubusercontent.com/arambazamba/az-204/main/Setup/create-lab-vm.azcli | bash
```



```
create-lab-vm.azcli :
```

```
rnd=$RANDOM
loc=westeurope
grp=az-lab
vmname=labvm-$rnd
user=azlabadmin
pwd=Lab@dm1n1234

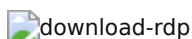
az group create -n $grp -l $loc

az vm create -g $grp -n $vmname --admin-username $user --admin-password $pwd --image
MicrosoftWindowsDesktop:Windows-10:win10-21h2-pro-g2:latest --size Standard_E2s_v3 -
-public-ip-sku Standard
```

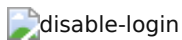
Connect to VM:

Go to Ressource Group `az-lab` and connect to VM using RDP and the credentials that you have used in the script:

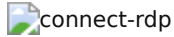
Download RDP File:



Optional - Disable Login:

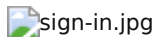


Sign In & Remember:

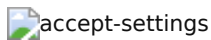


Credentials:

```
user=azlabadmin  
pwd=Lab@dm1n1234
```



Accept Settings:



Environment Setup, Debugging & State Management

This module demonstrates how to code & debug a Dapr based microservices as well as to deploy it to Azure Container Apps. It is based on the [Dapr quickstarts](#).

It contains two projects:


- [food-api-dapr](#) - A .NET Core Web API project that uses State Management to store and retrieve state. In other demos it will be used to demonstrate features like Secrets, Publish & Subscribe as well as Observability and Distributed tracing.
- [food-mvc-dapr](#) - A .NET MVC project that consumes the api using service invocation.
- [food-invoices-dapr](#) - A .NET Core Web API project that uses Publish & Subscribe to receive food orders, store them in a database and send an invoice to the customer.

Configuration of [Dapr components](#) is stored in the [components](#) folder of the apps base directory. During development it will use Redis as the default state store. When deploying it will use Azure Blob Storage. We could also use Azure Cosmos DB as a state store just by changing the state store configuration.

- `statestore.yaml` - Configures the state store to use Azure Blob Storage.

```
apiVersion: dapr.io/v1alpha1  
kind: Component  
metadata:  
  name: foodstore  
spec:  
  type: state.redis  
  version: v1  
metadata:
```

```
- name: redisHost
  value: localhost:6379
- name: redisPassword
  value: ""
```

 dapr-state

Docs & Resources

[Dapr Overview](#)

[Dapr CLI](#)

[Dapr Visual Studio Code extension](#)

[Developing Dapr applications with Dev Containers](#)

[Dapr on YouTube](#)

[eShopOnDapr](#)

Getting started, Basic State & Deployment to Azure Container Apps

Note: This demo assumes that you have created an Azure Container Registry and Azure Container Apps environment. If you haven't done so, please follow the [instructions](#) to provision the required Azure Resources using [Azure CLI](#) or [Bicep](#).

Dapr Environment Setup & Debugging

- Install Dapr CLI

```
Set-ExecutionPolicy RemoteSigned -scope CurrentUser
powershell -Command "iwr -useb
https://raw.githubusercontent.com/dapr/cli/master/install/install.ps1 | iex"
```

Note: Restart the terminal after installing the Dapr CLI

- Initialize default Dapr containers and check running containers:

```
dapr init
```

 dapr-init

Note: To remove the default Dapr containers run `dapr uninstall`

- Run project [food-api-dapr](#)

```
dapr run --app-id food-api --app-port 5000 --dapr-http-port 5010 --resources-
path './components' dotnet run
```


Note: By default the `--app-port` is launching the `https-profile` from `launchSettings.json`. With `.NET 7+` you can choose the profile by using the `--launch-profile` parameter.

- Test the API by invoking it several times using the `dapr` sidecar. The sidecar that listens to port `5010` forwards the request to the app. The sidecar is also responsible for service discovery and pub/sub.

```
GET http://localhost:<dapr-http-port>/v1.0/invoke/<app-id>/method/<method-name>
GET http://localhost:5010/v1.0/invoke/food-api/method/food
```

Note:

- Run project `food-mvc-dapr`

```
cd food-mvc-dapr
dapr run --app-id food-fronted --app-port 5002 --dapr-http-port 5020 --resources-path './components' dotnet run
```

- Show Dapr Dashboard

```
dapr dashboard
```

- Examine Dapr Dashboard on <http://localhost:8080>:



Running multiple microservices with Tye

- Install [Tye](#). Project Tye is an experimental developer tool that makes developing, testing, and deploying microservices and distributed applications easier

```
dotnet tool install -g Microsoft.Tye --version "0.11.0-alpha.22111.1"
```

- Create a `tye.yaml` file in the root of the solution by running:

```
tye init
```

Note: You can skip this step as the `tye.yaml` file is already included in the solution.

- A typical `tye` file could look like this:

```
name: dapr-services
services:
- name: food-api-dapr
project: food-api-dapr/food-api-dapr.csproj
bindings:
- port: 5000
- name: food-ui-dapr
project: food-ui-dapr/food-ui-dapr.csproj
```

```
bindings:
- port: 5002
```

- Run the two projects with Tye

```
tye run
```



Using Default State Store

- Add DaprClient to Program.cs

```
var builder = WebApplication.CreateBuilder(args);
...
// Add DaprClient to the ioc container
builder.Services.AddDaprClient();
```

- Examine CountController.cs and call getCount() multiple times to increment the counter and receive its current value:

```
public CountController(DaprClient daprClient)
{
    client = daprClient;
}

[HttpGet("getCount")]
public async Task<int> Get()
{
    var counter = await client.GetStateAsync<int>(storeName, key);
    await client.SaveStateAsync(storeName, key, counter + 1);
    return counter;
}
```

- To increment the counter you can use the pre-configured REST calls in [test-dapr.http](#) which is using the [Rest Client for Visual Studio Code Extension](#).


```
@baseUrl = http://localhost:5000
### Get the count and increment it by 1
GET {{baseUrl}}/count/getcount HTTP/1.1
```

- Check the state store data in the default state store - Redis:

```
dapr state list --store-name statestore
```

- Examine the Dapr Attach config in launch.json and use it to attach the debugger to the food-api-dapr process and debug the state store code:

```
{
  "name": "Dapr Attach",
  "type": "coreclr",
  "request": "attach",
  "processId": "${command:pickProcess}"
}
```

 filter-process

Deploy to Azure Container Apps

- Build the food-api-dapr image

```
env=dev
grp=az-native-$env
loc=westeurope
acr=aznative$env
imgBackend=food-api-dapr:v1
az acr build --image $imgBackend --registry $acr --file dockerfile .
```

- Create a storage account to be used as state store

```
stg=aznative$env
az storage account create -n $stg -g $grp -l $loc --sku Standard_LRS
```

- Update its values in `components/statestore-blob.yaml`

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: statestore
spec:
  type: state.azure.blobstorage
  metadata:
    - name: storageAccount
      value: aznative$env
    - name: storageAccessKey
      value: <storage-account-key>
```

- Add the Dapr component to the Azure Container Apps environment

```
az containerapp env dapr-component set -n $acaenv -g $grp \
--dapr-component-name statestore \
--yaml './components/statestore-blob.yaml'
```

Note. In Azure Portal you can also create the Dapr component in the Azure Container Apps environment. It allows you to choose between Redis, Azure Blob Storage, Azure Cosmos

DB and others as a state store. The interaction with the specifics of the state store is abstracted away by Dapr:



- Execute `deploy-app.azcli` to create the container app

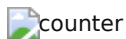
```
az containerapp create -n $appBackend -g $grp \  
--image $imgBackend \  
--environment $acaenv \  
--target-port 80 --ingress external \  
--min-replicas 0 --max-replicas 1 \  
--enable-dapr \  
--dapr-app-port 80 \  
--dapr-app-id $appBackend \  
--registry-server $loginSrv \  
--registry-username $acr \  
--registry-password $pwd
```

Note: Accessing ACR could also be done using a managed identity. Check the [documentation](#) for more details.

- Execute the `/count/getCount` method multiple times to increment the counter

```
curl -X GET "http://<URL>.$loc.azurecontainer.io/count/getCount" -H "accept: text/plain"
```

- Examine the storage account to see the state store data



Dapr Service Invocation, Pub / Sub & Bindings

- Dapr Service Invocation
- Dapr Pub/Sub
- Dapr Bindings

Dapr Service Invocation

- [food-api-dapr](#) is a REST API that exposes a set of endpoints to manage food items.

```
[HttpGet()]  
public IEnumerable<FoodItem> GetFood()  
{  
    return ctx.Food.ToArray();  
}
```

- This method is consumed by [food-mvc-dapr](#). Examine its [Program.cs](#) and notice the following code:

```
builder.Services.AddDapr();
...
app.UseCloudEvents();
...
app.MapSubscribeHandler();
```

- `AddDapr()` registers the necessary services to integrate Dapr into the MVC pipeline. It also registers a `DaprClient` instance into the dependency injection container.
- `UseCloudEvents()` adds `CloudEvents` middleware into the ASP.NET Core middleware pipeline. This middleware will unwrap requests that use the `CloudEvents` structured format, so the receiving method can read the event payload directly.
- `MapSubscribeHandler()` registers a route handler for the `dapr/subscribe` endpoint. This endpoint is used by Dapr to register the subscriber with the pub/sub component. The route handler will read the topic name from the request and register the subscriber with the pub/sub component.
- Examine the current state of [HomeController.cs](#) and notice that it is using direct service invocation to get the food items:

```
public async Task<IActionResult> Index()
{
    HttpClient client = new HttpClient();
    var daprResponse = await client.GetAsync($"http://localhost:
{BACKEND_PORT}/v1.0/invoke/{BACKEND_NAME}/method/food");
    var jsonFood = await daprResponse.Content.ReadAsStringAsync();
    ViewBag.Food = JsonSerializer.Deserialize<List<FoodItem>>(jsonFood);
    return View();
}
```

Note: The reason why we are not consuming the REST-API directly is because we want to use the benefits of Dapr service invocation which are:

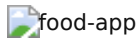
- Service discovery
- Standardizing API calls between services.
- Secure inter-service communication.
- Mitigating request timeouts or failures and automatic handling of retries and transient errors
- Implementing observability and tracing using OpenTelemetry

 dapr-service-invocation

- Run the UI and test the implementation:

```
cd food-dapr-fronted
dapr run --app-id food-fronted --app-port 5002 --dapr-http-port 5011 dotnet
run
```

- It should return the following result:

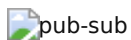


Dapr Pub/Sub

Dapr pub/sub building block provides a platform-agnostic API framework to send and receive messages. The publisher services publish messages to a named topic. Your consumer services subscribe to a topic to consume messages:

- pubsub.yaml :

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: food-pubsub
spec:
  type: pubsub.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      value: ""
```



Publisher

- Examine [FoodController.cs](#)

```
[HttpPost("add")]
public async Task<FoodItem> AddFood([FromBody] FoodItem food)
{
    logger.LogInformation("Started processing message with food name '{0}'",
        food.Name);
    var existing = ctx.Food.FirstOrDefault(f => f.ID == food.ID);
    if (existing != null)
    {
        ctx.Attach(food);
        ctx.Entry(food).State = EntityState.Modified;
    }
    else
    {

```

```

        ctx.Food.Add(food);
        logger.LogInformation("Food with ID '{0}' does not exist. Adding it",
            food.ID);
    }
    await ctx.SaveChangesAsync();
    await PublishFoodAdded(food);
    return food;
}

```

- Examine `PublishFoodAdded()` . It is responsible for publishing the food item to the Dapr Pub/Sub component:

```

private async Task PublishFoodAdded(FoodItem food)
{
    var pubsubName = cfg.GetValue<string>("PUBSUB_NAME");
    var topicName = cfg.GetValue<string>("PUBSUB_TOPIC");
    await client.PublishEventAsync(pubsubName, topicName, food);
}

```

Note: The `PublishEventAsync` method is used to publish the food item to the pub/sub component. `food-pubsub` is the name of the pub/sub component and `food-items` is the topic name.

- Run the api with Dapr and add the pub/sub component from the components folder:

```

dapr run --app-id food-fronted --app-port 5002 --dapr-http-port 5011 --
resources-path './components' dotnet watch run

```

Note: The `--resources-path` parameter is used to specify the location of the components folder. It adds all the components of the folder to the app. Previously we used `--components-path` to add components. In `tasks.json` this is still the case.

Subscriber

- [food-invoices-dapr](#) subscribes to the topic. Examine its [Program.cs](#) and notice the following code:

```

builder.Services.AddDapr();
...
app.UseCloudEvents();
...
app.MapSubscribeHandler();

```

- `AddDapr()` registers the necessary services to integrate Dapr into the MVC pipeline. It also registers a `DaprClient` instance into the dependency injection container.
- `UseCloudEvents()` adds `CloudEvents` middleware into the ASP.NET Core middleware pipeline. This middleware will unwrap requests that use the `CloudEvents` structured format, so the receiving method can read the event payload directly.

- `MapSubscribeHandler()` registers a route handler for the `dapr/subscribe` endpoint. This endpoint is used by Dapr to register the subscriber with the pub/sub component. The route handler will read the topic name from the request and register the subscriber with the pub/sub component.
- [InvoicesController](#) is responsible for receiving the food items and creating invoices:

```
[HttpPost]
[Dapr.Topic("food-pubsub", "food-items")]
public ActionResult CreateInvoice([FromBody] FoodItem food )
{
    // Create invoice
    ...
    return Ok("Invoice Created");
}
```

Note: The `Topic` attribute is used to register the subscriber with the pub/sub component. `food-items` is the topic name and `food-pubsub` is the name of the pub/sub component.

- To publish an item use:

```
POST http://localhost:5010/v1.0/publish/food-pubsub/food-items HTTP/1.1
content-type: application/json

{
  "id": 12,
  "name": "Pad Kra Pao",
  "price": 12.0,
  "inStock": 9,
  "pictureUrl": null,
  "code": "kra"
}
```

Dapr Bindings

Dapr bindings are a way to declaratively connect your application to another service. Dapr bindings are event-driven and can be triggered by an event or run on a schedule. Dapr bindings are implemented as an output binding, an input binding, or a bidirectional binding.

Tooling

- [Getting Started](#)
- [Azure Pass](#)
- [GitHub](#)
- [Visual Studio Code](#)
- [Markdown](#)
- [Azure CLI](#)
- [Docker and WSL Setup](#)
- [Configure VS Code REST Client Extension](#)